Model Checking Complete Requirements Specifications Using Abstraction[*]

Memorandum Report NRL/MR/5540–97–7999,
Naval Research Laboratory, Washington, DC 20375. Nov. 10, 1997

Ramesh Bharadwaj and Constance Heitmeyer
Center for High Assurance Computer Systems (Code 5546)
Naval Research Laboratory
Washington, DC 20375

Abstract

Although model checking has proven remarkably effective in detecting errors in hardware designs, its success in the analysis of software specifications has been quite limited. Model checking algorithms for hardware verification commonly use Binary Decision Diagrams (BDDs), a highly effective technique for analyzing specifications with the scores of Boolean variables commonly found in hardware descriptions. Unfortunately, BDDs are relatively ineffective for analyzing software specifications, which usually contain not only Booleans but variables spanning a wide range of data types. Further, software specifications have huge, often infinite, state spaces that cannot be model checked directly using conventional symbolic methods. One promising, but largely unexplored technique for limiting the size of the state space to be analyzed by model checking is to extract a model with a smaller state space from a complete specification using sound abstraction methods. Users of model checkers routinely analyze reduced models but most often generate the models in ad hoc ways. As a result, the reduced models are often incorrect.

This paper first describes how one can model check a complete requirements specification expressed in the SCR (Software Cost Reduction) tabular notation. Unlike previous approaches which applied model checking to mode transition tables with Boolean variables, we use model checking to analyze properties of a complete SCR specification with variables ranging over many data types. The paper also describes two sound and complete methods for producing abstractions from requirements specifications. These abstractions are derived from the specification based on the property to be analyzed. Finally, the paper describes how SCR requirements specifications can be translated into the languages of Spin, an explicit state model checker, and SMV, a symbolic model checker, and presents the results of model checking two sample SCR specifications using our abstraction methods and the two model checkers.

[*]This work was supported by the Office of Naval Research. This report is an extended version of the paper "Verifying SCR Requirements Specifications Using State Exploration," which appeared in *Proc., First ACM SIGPLAN Workshop on Automatic Analysis of Software*, Jan. 14, 1997.

1 Introduction

During the last decade, model checking has proven remarkably effective for detecting errors in hardware designs and protocols. Much of this success can be traced to the use of Binary Decision Diagrams (BDDs), an extremely efficient technique for symbolically representing Boolean formulae. Unfortunately, model checking has had only limited success in analyzing software specifications, largely because software specifications routinely contain not only Booleans but variables spanning a wide range of data types, including integers, reals, and enumerated types. A further barrier is the huge, in many cases infinite, state spaces that must be analyzed in model checking software specifications. Due to these very large state spaces and the rich data types commonly found in software specifications, BDD-based model checkers have proven relatively ineffective.

Before practical software specifications can be analyzed efficiently by model checking, the *state explosion problem* must be addressed, i.e., the size of the state space to be analyzed must be reduced. The current users of model checkers generate such reductions routinely but almost always in ad hoc ways [24]: the correspondence between the reduced models and the original specifications is based on informal, intuitive arguments. One consequence of this informal process is that the models analyzed by model checkers are often incorrect. For example, when Dill et al. analyzed the errors detected by their model checker Murphi, they found that valid design errors were very rare, whereas human errors in translating the original design to the model analyzed by Murphi were frequent [38]. Hence, a serious problem is that reduced models generated informally and by hand may not be true abstractions of the original design.

In contrast, our approach derives the abstract models systematically from the requirements specification and the formula to be analyzed. Users of our methods need not design the abstractions; instead, the abstractions can be derived automatically. With our approach, analyzing a specification for errors consists of three steps. First, our abstraction methods are used to produce an abstract model. Next, a model checker is executed to analyze the abstract model for the property of interest. In the third step (required when the model checker detects a violation of the property), the counterexample produced by the model checker is translated to a corresponding counterexample in the original specification. This last step is crucial because the user's understanding of the system will be in terms of the original specification rather than the abstract model.

Our abstraction methods are designed for specifications expressed in a tabular notation called SCR (Software Cost Reduction). For a number of years, researchers at the Naval Research Laboratory have been developing a formal method based on the SCR notation to specify the requirements of computer systems [19, 1]. The SCR method, originally formulated to document the requirements of the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft, was introduced more than 15 years ago. Since then, many industrial organizations, including Bell Laboratories [20], Grumman [32], and Ontario Hydro [34], have used the SCR method to specify requirements. Recently, a version of the SCR method called CoRE [11] was used to document the requirements of Lockheed's C-130J Operational Flight Program (OFP) [12]. The OFP consists of more than 230K lines of Ada code [39], thus demonstrating the scalability of SCR.

We have developed a formal state machine model to define the SCR semantics [18, 14] and a set of formal techniques and software tools to analyze requirements specifications in the SCR notation [15, 16, 14]. The tools include a *specification editor* for creating and modifying a requirements specification, a *consistency checker* which checks the specification for well-formedness (e.g., syntax and type correctness, no missing cases, no circular definitions, and no unwanted nondeterminism), and a *simulator* for symbolically executing the specification to ensure that it captures the customer's intent. Recently, we added a model checking capability to the toolset. Once the user has developed and refined an SCR requirements specification with our tools, he can invoke the Spin model checker [21, 22] within the toolset to analyze a specification for application properties. To make model checking feasible, the user can apply our abstraction methods to the specification prior to invoking Spin.

An early application of model checking to SCR requirements specifications was reported in 1993 by Atlee and Gannon, who used the model checker MCB [8] to analyze properties of individual mode transition tables taken from SCR specifications [4]. More recently, Sreemani and Atlee [37] used the symbolic model checker SMV [31] to determine whether the mode transition tables in the original A-7 requirements document satisfied assertions about combinations of modes. The latter experiment demonstrates that model checking can analyze requirements specifications of moderate size.

A major goal of our work is to generalize and extend some aspects of the earlier techniques for model checking SCR requirements specifications. While the techniques of Atlee et al. are designed to analyze properties of mode transition tables with Boolean input variables, the approach we describe can be used to analyze properties of a complete SCR specification: The properties to be analyzed can contain *any* variable in the specification, and we allow variables to range over varied domains, such as integer subranges, enumerated values, and infinite subranges of the real numbers.

This paper makes the following contributions to the model checking of software specifications:

- In Section 3, a method is described for model checking complete SCR specifications rather than individual mode transition tables. The variables in the specification can have varied types—e.g., Boolean, enumerated, integer, or real.

- In Section 4, two methods are presented for deriving abstractions from SCR specifications. These abstractions are derived automatically from the formula to be analyzed and the SCR specification.

- In Section 5, methods are presented for translating an SCR specification into both *Promela*, the language of Spin, and the language of SMV.

Finally, Section 6 summarizes the results of our experiments with Spin and SMV, Section 7 discusses related work, and Section 8 describes our ongoing and future work.

2 Background

2.1 SCR Requirements Model

An SCR requirements specification describes a system as a composition of a nondeterministic environment and a (usually) deterministic system [18]. The system environment contains *monitored quantities*, environmental quantities that the system monitors, and *controlled quantities*, environmental quantities that the system controls. The environment nondeterministically produces a sequence of input events, where an *input event* signals a change in some monitored quantity. The system, which is represented in our model as a state machine, begins execution in some initial state. It responds to each input event in turn by changing state and by producing zero or more system outputs, where a *system output* is a change in a controlled quantity. In SCR, we assume that the system behavior is synchronous (similar to Esterel's Synchrony Hypothesis [5]), that is, the system completely processes one input event before the next input event is processed.

In our requirements model, a system Σ is represented as a 4-tuple, $\Sigma = (S, S_0, E^m, T)$, where S is a set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is the transform describing the allowed state transitions [18]. In the initial version of our formal model, the transform T is deterministic, i.e., a function that maps an input event and the current state to a new state. The transform T is the composition of smaller functions, called *table functions*, which are derived from the tables in an SCR requirements specification. Our formal model requires the information in each table to satisfy certain properties. These properties guarantee that each table describes a total function.

In SCR, the required system behavior is described by NAT and REQ, two relations, of the Parnas-Madey Four Variable Model [35]. NAT describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment. REQ describes the required relation between the monitored and the controlled variables. To specify REQ concisely, the SCR approach uses four constructs – mode classes, terms, conditions, and events. A *mode class* is a variable whose values are *system modes* (or simply *modes*), while a *term* is any function of monitored variables, modes, or other terms. A *variable* is any monitored or controlled variable, mode class, or term. The SCR requirements model includes a set $RF = \{r_1, r_2, \ldots, r_n\}$ containing the names of all variables in a given specification, and a function TY which maps each variable to the set of its legal values. In the model, a *state* s is a function that maps each variable in RF to its value, a *condition* is a predicate defined on a system state, and an *event* is a predicate defined on two system states. We say an event "occurs" when the value of any system variable changes. The notation "`@T(c) WHEN d`" denotes a *conditioned event*, defined as

$$\texttt{@T(c) WHEN d} \overset{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed conditions c and d are evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state.

To compute the new state, the transform T uses the values of variables in both the old state and the new state. To describe the variables on which a given variable "directly depends" in the new state, we define *dependency relations* D_{new}, D_{old}, and D on $\mathrm{RF} \times \mathrm{RF}$. For variables r_i and r_j, the pair $(r_i, r_j) \in D_{new}$ if r'_j is a parameter of the function defining r'_i; the pair $(r_i, r_j) \in D_{old}$ if r_j is a parameter of the function defining r'_i; and $D = D_{new} \cup D_{old}$. To avoid circular definitions, we require D_{new} to define a partial order. Because they depend only on changes in the environment, the monitored variables are first in the partial order. Because they can depend on any monitored variable, term, or mode class, the controlled variables come last in the partial order. The mode classes and terms come between the monitored and controlled variables. The assumptions that the table functions are total and that the variables in RF are partially ordered guarantee that the transform T is a *function* (at most one new system state is defined) and *well-defined* (for each enabled input event, at least one new system state is completely defined) [14, 18].

2.2 Types and Dependencies Sets

To illustrate the SCR constructs, we consider a simplified version of a control system for safety injection [9]. Appendix A contains a prose description of the behavior of this system, three tables taken from an SCR specification of the required system behavior, and the table functions that can be derived from the tables using our formal model. In the example system, the set of variable names RF contains the three monitored variables `Block`, `Reset`, and `WaterPres`, the mode class `Pressure`, the term `Overridden`, and the controlled variable `SafetyInjection`. The type definitions include

$$\begin{aligned}
&\mathrm{TY}(\texttt{Block}) = \{\texttt{On, Off}\}\\
&\mathrm{TY}(\texttt{Reset}) = \{\texttt{On, Off}\}\\
&\mathrm{TY}(\texttt{SafetyInjection}) = \{\texttt{On, Off}\}\\
&\mathrm{TY}(\texttt{Pressure}) = \{\texttt{TooLow, Permitted, High}\}\\
&\mathrm{TY}(\texttt{Overridden}) = \{\textit{true, false}\}
\end{aligned}$$

(The type definition of `WaterPres`, which has a large range of possible values, is given in Section 2.3.) The new state dependency relation D_{new} for the example system is

$$\begin{aligned}
&\{(\texttt{SafetyInjection}, \texttt{Pressure}),\\
&\ (\texttt{SafetyInjection}, \texttt{Overridden}),\\
&\ (\texttt{Pressure}, \texttt{WaterPres}), (\texttt{Overridden}, \texttt{Pressure}),\\
&\ (\texttt{Overridden}, \texttt{Block}), (\texttt{Overridden}, \texttt{Reset})\}.
\end{aligned}$$

The partial order defined by D_{new} is

$$< (\texttt{Block}, \texttt{Reset}, \texttt{WaterPres}), \texttt{Pressure}, \texttt{Overridden}, \texttt{SafetyInjection} > .$$

2.3 Models of the Monitored Variables

As noted above, the system behavior described by our model has a nondeterministic part and a deterministic part. While the transform T is deterministic, the input events, which are produced by the environment, are nondeterministic. The monitored variables involved in the input events may each be represented as simple finite state machines with an initial state, a set of possible states (defined by the function TY), and a next-state relation. For example, the monitored variables `Block` and `Reset` in the sample system both have `Off` as the initial state, the set {`Off`, `On`} as the possible states, and the set {(`Off`, `On`), (`On`, `Off`)} as the next-state relation. One possible model of the monitored variable `WaterPres` has an initial state of 14, possible values defined by TY(`WaterPres`) = $\{0, 1, 2, \ldots, 2000\}$, and a next-state relation τ_{wp}, which allows `WaterPres` to change by at most 10 units from one state to the next, i.e.,

$$\tau_{wp} = \{(x, x') : 1 \leq |x' - x| \leq 10,\ 0 \leq x \leq 2000,\ 0 \leq x' \leq 2000\}. \qquad (1)$$

An important assumption of our model, the One Input Assumption, states that only one monitored variable changes at each state transition. Using the above models of the monitored variables, we can show that when the sample system is in its initial state, all of the following input events are enabled: @T(`Block=On`), @T(`Reset=On`), and @T(`WaterPres=`x), where $4 \leq x \leq 24$ and $x \neq 14$. The One Input Assumption allows exactly one of these input events to occur at the next state transition.

3 Verifying SCR Specifications

This section describes our use of model checking for verification and error detection. By verification, we mean the process of establishing logical properties of an SCR specification. We specify the properties as logical formulae. In this paper, we focus on a class of properties known as *state invariants*. Each invariant may be either a *one-state* invariant, a property of every reachable system state $s \in S$, or a *two-state* invariant (also called a *transition* invariant), a property of every pair of reachable states (s, s'), where $s, s' \in S$ and there exists an enabled input event $e \in E^m$ such that $T(e, s) = s'$. We focus on one-state and two-state invariants because they are the properties most commonly found in specifications of practical systems we have studied (e.g., the A-7 OFP [1], Kirby's cruise control system [28], and, most recently, a safety-critical component of a Navy system).

In the following, we assume that a given SCR specification satisfies application-independent properties – that is, the specification is type correct, the table functions derived from the specification are total functions, etc. Such properties can be established using our toolset. For details of how these checks are carried out, see [14].

To demonstrate the properties we would like to establish, we consider the following properties for the safety injection specification:

1. `Reset` $=$ `On` $\wedge$ `Pressure` $\neq$ `High` $\Rightarrow$ $\neg$`Overridden`
2. `Reset` $=$ `On` $\wedge$ `Pressure` $=$ `TooLow` $\Rightarrow$ `SafetyInjection` $=$ `On`
3. `Block` $=$ `Off` $\wedge$ `Pressure` $=$ `TooLow` $\Rightarrow$ `SafetyInjection` $=$ `On`
4. @T(`Pressure` $=$ `TooLow`) WHEN `Block` $=$ `Off` $\Rightarrow$ `SafetyInjection`$'$ $=$ `On`

The first three properties are all one-state invariants. The fourth property, a two-state invariant, states, "*If* `Pressure` *becomes* `TooLow` *in the new state and* `Block` *is* `Off` *in the old state, then* `SafetyInjection` *is* `On` *in the new state.*"

To establish a formula q as a one-state invariant (two-state invariant) of a state machine, we need to show that q holds in every reachable state (every reachable transition-pair of states) of the machine. We do this by starting from the initial state and repeatedly computing the next states until a fixpoint is reached. To compute the possible new states given a current state, we need representations of the transform T and of the input events that trigger the state transitions. Recall that the system transform T (which specifies the new values of system variables) is deterministic, whereas the state machines for monitored variables are nondeterministic. The nondeterminism has two aspects: (a) at a given step, many monitored quantities are eligible to change (i.e., are enabled) and (b) at a given step, a monitored variable may change in more than one way.

3.1 Conditional Assignment

To compute the next states, we associate with each variable r_i in RF, a conditional assignment of the form:

```
if
      □  $g_{i,1} \rightarrow r_i := v_{i,1}$
      □  $g_{i,2} \rightarrow r_i := v_{i,2}$
⋮
      □  $g_{i,n_i} \rightarrow r_i := v_{i,n_i}$
fi
```

Here, $g_{i,1}, g_{i,2}, \ldots, g_{i,n_i}$ are boolean expressions (guards) and $v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}$ are expressions that are type compatible with variable r_i. We define the semantics of a conditional assignment along the lines of the enumerated assignment of UNITY [6] – one assignment whose associated guard is "true" is executed at each transition. If more than one guard is "true", then any one of the associated assignments is nondeterministically chosen. If no guard is "true", the variable value is left unchanged. To represent the functions defined by SCR tables, we allow the expressions $g_{i,1}, g_{i,2}, \ldots, g_{i,n_i}$ and $v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}$ to refer to both "old" and "new" values of variables, provided that the "new" references are not circular.

For the control system example, the conditional assignment for the term `Overridden` is given below. Conditional assignments for variables `Pressure` and `SafetyInjection` can be expressed in a similar fashion.

```
if
     □ @T(Block=On) AND (Pressure=TooLow) AND
        (Reset=Off) -> Overridden := true
     □ @T(Block=On) AND (Pressure=Permitted) AND
        (Reset=Off) -> Overridden := true
     □ @T(Pressure=High) -> Overridden := false
     □ @T((Pressure=TooLow) OR (Pressure=Permitted))
        -> Overridden := false
     □ @T(Reset=On) AND (Pressure=TooLow)
        -> Overridden := false
     □ @T(Reset=On) AND (Pressure=Permitted)
        -> Overridden := false
fi
```

The following is the conditional assignment for the monitored variable `Block`. Conditional assignments for the monitored variables `WaterPres` and `Reset` can be expressed in a similar fashion.

```
if
     □  (Block=Off)  ->  Block := On
     □  (Block=On)   ->  Block := Off
fi
```

Note that if `Block` or `Reset` change at a given step, each can only change in one way. In contrast, if `WaterPres` changes, it may change in many ways.

3.2 Computing the New State

Given a current state s and the conditional assignments for all monitored variables, we can determine the set of input events that are enabled in s by evaluating each guard. Each guard that evaluates to "true" along with the associated assignment determines an input event that is enabled in s. Because the One Input Assumption only allows a single input event to occur at each transition, one of the enabled input events e is selected nondeterministically.

The selected input event e and the current state s determine the new state s'. The values of the monitored variables in the new state s' are determined solely by the input event e. The values of the other variables in the new state s' (the mode classes, the terms, and the controlled variables) can be computed from the conditional assignments for these variables. The partial order determines the sequence in which the conditional assignments are evaluated. Because a total function defines the value of each mode class, term, and controlled variable, exactly one guard of each conditional assignment will evaluate to "true" and exactly one assignment can be executed per variable.

4 Two Abstraction Methods

By their very nature, the number of reachable states in practical systems is usually very large in relation to their logical representation. Hence, for realistic software specifications, most fixpoint computations fail to terminate because they run out of memory. Several techniques have been proposed to combat state explosion in model checking. One approach proposed by Clarke et al. in 1994 is to use abstraction [7]. Although abstraction could theoretically reduce a huge (and even infinite) state space to a much smaller state space, practical ways of deriving abstractions have not emerged. In fact, the use of abstraction in model checking is widely regarded as impractical (see, e.g., [25]).

Below, we describe two methods for deriving abstractions from SCR requirements specifications based on the formula to be analyzed. Both methods are practical: Neither requires ingenuity on the user's part, and each derives a smaller, more abstract model automatically. Further, each method systematizes techniques that current users of model checkers routinely apply but in ad hoc ways.

Applying our methods eliminates certain variables and their associated tables from the full SCR specification. Instead of model checking the full SCR specification of the state machine Σ for the property q, we model check an abstract SCR machine Σ_A for a corresponding property q_A. (The abstract property q_A is syntactically identical to property q, but because it is defined over a projection of the domain over which q is defined, we call it q_A.) Our abstraction methods are both sound and complete. Given property q and state machine Σ, we say that Σ_A is a *sound* abstraction of Σ relative to q if q_A is an invariant of Σ_A implies that q is an invariant of Σ. Given property q and state machine Σ, we say that Σ_A is a *complete* abstraction of Σ relative to q if q is an invariant of Σ implies that q_A is an invariant of Σ_A. Completeness is an especially desirable property in model checking. Because most practical specifications are much too large to analyze completely, the most useful function of model checking is to detect errors. Detecting an error when the abstraction is complete means that any counterexample detected in the abstraction corresponds to a counterexample in the original specification.

To characterize the allowed state transitions of Σ below, we define a next-state predicate ρ on pairs of states such that $\rho(s, s')$ is true iff there exists an enabled event $e \in E^m$ such that $T(e, s) = s'$. The predicate ρ is simply a concise and abstract way of expressing the transform T. The corresponding next-state predicate for Σ_A is ρ_A.

4.1 Abstraction Method 1: Eliminate Irrelevant Entities

This method uses the set of variable names which occur in the formula being analyzed to eliminate unneeded variables and the tables that define them (and the state machines in the case of monitored variables) from the analysis. To apply this method, we identify the set $\mathcal{O} \subseteq RF$ of variables occurring in formula q. Then, we let set $\mathcal{O}^*$ be the reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D of an SCR specification for state machine Σ. It is sound to infer the invariance of q for Σ if q_A is an invariant of the abstract machine Σ_A with $RF_A = \mathcal{O}^*$ and if the system transform of Σ_A, ρ_A, is obtained from ρ by deleting all associated tables (or state machines in the case of monitored variables) for variables in the set $RF - RF_A$. This abstraction method is also complete. Therefore, we always apply this abstraction method automatically before every verification.

For example, suppose we are analyzing the invariance of property 1 in Section 3 for the safety injection system. We identify the set of variables $\mathcal{O}$ occurring in the formula as

$$\mathcal{O} = \{\texttt{Pressure}, \texttt{Overridden}, \texttt{Reset}\}.$$

The reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D for safety injection is $\mathcal{O}^*$, which is defined by

$$\mathcal{O}^* = \{\texttt{Pressure}, \texttt{Overridden}, \texttt{Reset}, \texttt{Block}, \texttt{WaterPres}\}.$$

Applying this abstraction method eliminates the controlled variable `SafetyInjection`, together with its table, from the specification of the state machine Σ. The reduced specification describes the abstract machine Σ_A. Then, given an SCR specification of the state machine Σ and the property q, model checking the abstract machine Σ_A for property q_A is equivalent to model checking the original machine Σ for property q (in this case, property 1).

Applying this method can significantly reduce the size of the state space to be model checked. Suppose that variable r_i can be eliminated by this abstraction method and that the cardinality of r_i's type set is k_i. Then, the size of the state space that needs to be analyzed can be reduced by as much as a factor of k_i. If k_i is large, the amount of memory required to model check Σ_A may be considerably smaller than the amount of memory needed to model check Σ. Further, in many cases, many variables may be eliminated by this method and hence the savings in memory can be considerable, especially if some of the variables have large type sets.

4.2 Abstraction Method 2: Abstract Monitored Variables

Suppose that r is an monitored variable that does not appear in the formula q and that $\hat{r}$ is the only variable that depends on r. We define the set of variables of the abstract machine as $RF_A = RF - \{r\}$. That is, we simply remove r from the set of variables. In Σ_A, the dependence of $\hat{r}$ on r is eliminated by treating $\hat{r}$ as a monitored variable. The initial state(s), the set of possible states, and the next-state relation for the new monitored variable can be computed from $\hat{r}$'s initial state and from the table in Σ defining $\hat{r}$. We can generalize this method to eliminate many input variables $r_1, r_2, \ldots, r_n$ from RF. This reduction can be performed if $\hat{r}$ is the only variable that depends on $r_1, r_2, \ldots, r_n$ and if none of the variables $r_1, r_2, \ldots, r_n$ appear in q.

To illustrate this abstraction method, we consider the safety injection system. The root cause of state explosion when model checking this system is monitored variable `WaterPres`. We therefore wish to eliminate this monitored variable. Studying the specification of the safety injection system in Appendix A reveals that `WaterPres` only appears in Table 3, the table defining the mode class `Pressure`. Since `WaterPres` does not occur in any of the properties 1-4, nor in the tables for variables `Overridden` and `SafetyInjection`, we may delete `WaterPres`, and the table for variable `Pressure` when model checking properties 1-4. In constructing Σ_A, we define `Pressure` as a monitored variable with initial state `TooLow` and the set $\{\texttt{TooLow}, \texttt{Permitted}, \texttt{High}\}$ as the possible states. The next-state relation for `Pressure`, namely,

$$\{(\texttt{TooLow}, \texttt{Permitted}), (\texttt{High}, \texttt{Permitted}), (\texttt{Permitted}, \texttt{High}), (\texttt{Permitted}, \texttt{TooLow})\},$$

can be computed from the table defining `Pressure`.

In this abstraction method, the values of the detailed variable r are organized into equivalence classes by the abstract variable $\hat{r}$. In the safety injection example, we can define a function h that maps the type set of `WaterPres` onto the type set of `Pressure`; that is, h is a mapping from the set $\{0, 1, 2, \ldots, 2000\}$ onto the set $\{\texttt{TooLow}, \texttt{Permitted}, \texttt{High}\}$. The mapping h is defined by

$$h(\texttt{WaterPres}) = \begin{cases} \texttt{TooLow} & \text{if } \texttt{WaterPres} < \texttt{Low} \\ \texttt{Permitted} & \text{if } \texttt{WaterPres} \geq \texttt{Low} \wedge \texttt{WaterPres} < \texttt{Permit} \\ \texttt{High} & \text{if } \texttt{WaterPres} \geq \texttt{Permit} \end{cases}$$

(The constants `Low` and `Permit` are defined in Appendix A.) Clearly, h partitions the values of `WaterPres` into three equivalence classes, one corresponding to each of the three possible values of `Pressure`.

Because q is constant on every equivalence class, it is easy to see that this abstraction method is sound. Given some mild restrictions on the relationship between two states in the same equivalence class, this abstraction method is also complete. A sufficient condition for completeness is that any state s in an equivalence class must be reachable in a finite number of steps from any other state $\tilde{s}$ in the equivalence class. In the classes of systems we model, this condition can usually be satisfied. For example, in the safety injection system, if the abstract machine Σ_A is in its initial state, then the variable `Pressure` has the value `Low`. Consider a step in the abstract machine triggered by a change in `Pressure` from `Low` to `Permitted`. Starting in the initial state, the original machine Σ, which can only change `WaterPres` by at most 10 units from one state to the next, will require many steps (at least 88!) to reach the `Permitted` range. The definition of `WaterPres`'s next-state relation, τ_{wp} (see (1) in Section 2.2) guarantees that the above condition is satisfied. That is, given any two values $x, \tilde{x}$ of `WaterPres` in the same equivalence class, it is possible in a finite number of steps for `WaterPres` to transition from x to $\tilde{x}$.

5 Model Checking SCR Specifications.

This section describes how SCR requirements specifications can be translated into the languages of two model checkers—the explicit state model checker Spin and the symbolic model checker SMV.

5.1 Using the Spin Verifier

Spin [21, 22] is a model checker which uses state exploration for verifying properties. Systems are described in a language called *Promela* [21] and properties are expressed in linear-time temporal logic (LTL) [30]. Spin has been largely used to verify communication protocols and asynchronous hardware designs.

Promela, the language of Spin, is a notation loosely based on Dijkstra's "guarded commands" [10]. Supported data types in *Promela* include `bool` (booleans), `byte` (short unsigned integers), and `int` (signed integers). Control statements include the assignment statement, statement `skip` (which does nothing), sequential composition of statements, the conditional statement, and the iterative statement. The language also has an `assert` statement.

Translating an SCR specification to *Promela* proceeds as follows. Because *Promela* does not allow expressions containing both "old" and "new" values of variables, we assign two *Promela* variables to each variable in the SCR specification. We call these the "new" and "old" variables. Further, expressions containing the event notation `@T(c)` are translated into equivalent forms involving the "old" and the "new" variables. We translate the conditional assignment for each table into a *Promela* conditional statement, which computes the value of the "new" variable at each step. The conditional statements are executed sequentially, in a predetermined order consistent with the partial order induced by the new state dependency relation of the SCR specification. After all conditional assignments for table functions are executed and new values assigned to all "new" variables, all "old" variables are assigned their corresponding "new" values.

Further, we perform an optimization based on the fact that the system transform of an SCR specification is a function. This ensures that all conditional statements for variables other than the monitored variables are *deterministic*. Therefore, once we have selected an input event, we may compute the new state in a single step. In *Promela* we specify this by enclosing all the statements which correspond to the computation of the mode variables, the terms, and the controlled variables in a `d_step` (deterministic step) construct. This ensures that, for each input event, only one state (i.e., the new state) is entered into the hash table which stores the reachable states.

To generate *Promela* code corresponding to input events, we generate a nondeterministic conditional statement for each monitored variable, which assigns any value in the variable's domain to the "new" *Promela* variable. We "build in" the One Input Assumption by embedding all assignments to monitored variables in a single (nondeterministic) conditional statement. Appendix B presents the *Promela* code generated by the SCR* toolset (edited to enhance readability) for the safety injection example.

To check a one-state invariant with Spin, we embed the invariant in a *Promela* `assert` statement. Then, Spin checks the truth of the invariant in the initial state and in each generated "new" state. To check a two-state invariant, one could express the invariant in LTL and invoke the built-in translator of Spin to construct an equivalent `never` automaton. Since an SCR variable's "new" and "old" values are explicitly assigned to two *Promela* variables, we avoid this automaton construction for two-state invariants; instead, we check them directly in an `assert` statement. An advantage of this approach is that invariant checking in Spin is computationally more efficient than checking properties expressed as `never` automata.

To combat state explosion, conventional partial order reduction methods avoid the exploration of redundant interleavings by computing and keeping track of information about redundant interleavings *during* state exploration [40, 13, 23]. In our approach, it is sufficient to evaluate the next state using *only one* predetermined interleaving consistent with the partial order induced by the new state dependency relation. This property is common to all SCR specifications [18]. Therefore, enabling Spin's partial order reduction algorithm will almost never reduce the space requirement and may actually *increase* the required analysis time due to additional overhead.

5.2 Using the SMV Model Checker

SMV [31] is a tool for verifying properties of system descriptions expressed in a special-purpose language (also called SMV). Properties are expressed in the branching time temporal logic CTL. A system is described in SMV as a set of initial states and a transition (next-state) relation. Users may either use predicate logic or a more restricted description language to specify the transition relation. Although predicate logic provides considerable flexibility, its use can lead to inconsistency – if a formula specifying the transition relation is a logical contradiction, many properties will be vacuously true. Using only the restricted description language of SMV, which has a parallel assignment syntax similar to the notation of SCR, avoids this problem. For specifications in the restricted language, SMV checks for multiple parallel assignments to a variable, circular definitions, and type errors. These checks help ensure that all specifications in the restricted syntax are consistent.

Our translation of SCR specifications into SMV uses the restricted language of SMV. Our translation method is similar to that of Atlee et al. [3] but differs in two important ways. First, as mentioned above, we translate complete SCR specifications comprising monitored and controlled variables, mode classes, and terms, any of which may be of type Boolean, an integer subrange, or an enumeration. In contrast, Atlee et al. translate mode transition tables with Boolean input variables into SMV. Second, to check properties on two states, Atlee et al. introduce additional SMV variables, which record values of state variables in the previous state. In contrast, we encode two-state properties directly into CTL, using the algorithm of Jeffords [26]. This reduces the number of state variables in half; therefore, our method can potentially reduce the memory requirements for model checking by an exponential factor.

To translate an SCR specification into SMV, we express the conditional assignment corresponding to each table as an SMV `case` statement. This computes the value of the corresponding variable in the "new" state. Because the system transform T of an SCR specification is deterministic, the guards of all conditional assignments corresponding to system variables (i.e., variables other than monitored variables) are mutually disjoint. For these variables, it is straightforward to translate a conditional assignment into an SMV `case` statement. Note that in SMV a primed occurrence x' of a variable x is denoted by `next(x)`. Thus, the conditional assignment for `Overridden` is translated into the following SMV construct:

```
next(Overridden) :=
 case
    (next(Block) = On & Block = Off &
     Pressure = TooLow & Reset = Off) |
    (next(Block) = On & Block = Off &
     Pressure = Permitted & Reset = Off) : TRUE;
    (next(Reset) = On & Reset = Off &
     Pressure = TooLow) |
    (next(Reset) = On & Reset = Off &
     Pressure = Permitted) |
    (next(Pressure) = High & !(Pressure = High)) |
    ((next(Pressure) = Permitted | next(Pressure) = TooLow) &
    !(Pressure = Permitted | Pressure = TooLow)) : FALSE;
    1: Overridden; -- This means "otherwise Overridden"
 esac;
```

Unfortunately, when more than one guard is "true", the semantics of the SMV `case` statement differs from that of the enumerated assignment statement discussed in Section 3.1. In SMV, the first assignment whose guard is "true" is chosen, rather than a nondeterministic choice of any one of the assignments whose guards are "true". Therefore, when the guards of a conditional assignment are not mutually disjoint, a straightforward translation would be incorrect. We model nondeterministic choice by an explicit assignment of an arbitrary element among those in a set, using the SMV syntax of set assignment which denotes this operation.

For example, in the safety injection system, Abstraction Method 2 may be used to eliminate monitored variable `WaterPres` (see Section 4.2). The next-state relation for the mode class `Pressure` may be expressed in SMV as follows:

```
next(Pressure) :=
 case
    Pressure = Permitted : {TooLow, High};
    Pressure = TooLow    : Permitted;
    Pressure = High      : Permitted;
 esac;
```

We generate a nondeterministic SMV assignment that corresponds to the conditional assignment of each monitored variable, and a deterministic assignment for each term, mode class, and controlled variable. Unlike Spin, which executes conditional assignments sequentially, SMV performs all assignments in parallel, i.e., in "one step". Therefore, unlike the *Promela* model, the assignments in SMV may be ordered arbitrarily.

Unlike the translation to *Promela*, where we build the One Input Assumption and other restrictions imposed by NAT into the model, we encode NAT restrictions in SMV as predicates in a "`TRANS`" section. For example, the encoding of the state machine for `WaterPres` has two parts – the assignment statement for `WaterPres` allows *any* value from its domain for the new state (irrespective of WaterPres in the old state), while the predicate in the `TRANS` section restricts this change to be at most 10 units. Appendix C presents the SMV code generated by this method for the safety injection example.

5.3 Spin vs SMV

The relative merits of explicit state (also called concrete) model checkers, such as Spin, and "symbolic" model checkers, such as SMV, has sparked considerable controversy. Explicit model checkers compute the set of reachable states by enumeration (i.e., by "running the model"), whereas symbolic model checkers execute the model symbolically by representing the set of reachable states as a logical formula using a BDD. The state spaces of some hardware designs with a certain regularity in their structure have been shown to have very compact BDD representations. For such systems, the space requirement using BDDs has a linear, rather than an exponential, relationship with the number of state variables in the model. However, BDDs do "blow up" (i.e., have an exponential space requirement) when the models are more irregular, which is often the case in software specifications. Explicit state model checkers generally do better than BDDs on descriptions of communication protocols and control systems. This is because the space requirement of explicit state model checking is proportional not to the number of *possible* states (as in BDDs) but to the number of *reachable* states, which are far fewer for such systems. Not surprisingly, therefore, algorithms for explicit state enumeration seem to require less space than symbolic algorithms when model checking SCR specifications.

We note that it is possible to construct an explicit state model checker with an exponentially smaller memory requirement than the memory required by Spin in our experiments. In using Spin, we were forced to declare two *Promela* variables for each SCR variable with a potentially exponential increase in space requirements. (This limitation of *Promela* has nothing to do with the explicit state enumeration algorithms of Spin.) Because the SMV language allows references to both the "old" and "new" values of a variable, our SMV models avoided this problem. However, in SMV, there is a potential for exponential BDD blowups for specifications containing integer variables (such as the original Safety Injection specification) because SMV does not handle integer variables optimally [2]. As Anderson et al. show [2], a more optimal encoding for integer variables in SMV is possible and may produce exponential reductions in space requirements.

Symbolic model checkers such as SMV provide counterexamples for two-state properties as a linear trace, which may be difficult to interpret because the property is expressed in CTL, a branching time logic. However, symbolic model checkers have a distinct advantage over state enumeration in one respect: Expressing constraints (such as environmental restrictions on monitored variables) symbolically as logical formulae is more convenient than representing them as state machines. It is natural to allow, and efficient to implement, constraints expressed as logical formulae in symbolic model checkers. SMV allows users to intermix predicate logic with the more restrictive descriptive language. However, because this opens up the possibility of specifying logical contradictions, this feature should be used with caution.

Another advantage of SMV over Spin is that, when a property violation is detected, SMV is guaranteed to produce the shortest possible counterexample. Spin provides an algorithm that finds short counterexamples, but the algorithm does not always find the shortest one. This is because the symbolic model checking algorithms of SMV perform a breadth-first search of the state space in contrast to the depth-first search performed by the algorithms of Spin. Explicit state model checkers that perform a breadth-first search do exist; for example, the explicit state model checker Murphi implements a breadth-first search. However, algorithms used to generate counterexamples in symbolic model checking and explicit state enumeration by breadth-first search are considerably more expensive and complex than the corresponding algorithm for explicit state enumeration by depth-first search.

For many practical problems, a complete search of the state space (using either explicit state or symbolic methods) is usually infeasible. In such situations, model checking remains useful for *error detection*. Generally, we have found that explicit state methods are computationally less expensive than symbolic model checking for error detection, especially in cases where the specification is incorrect. The next section provides details.

6 Experimental Results

This section presents and discusses some results of our experiments with Spin and SMV. To evaluate the abstraction methods described above, we have applied them to several small examples and to a more realistic SCR specification.

For the safety injection specification (SIS), we were able to establish properties 1 and 2. We were also able to show that properties 3 and 4 are *not* invariants of the specification. One of the major problems in using model checking to evaluate abstract models is that counterexamples, which are generated in terms of the abstractions, are often hard to interpret (see, e.g., [36]). We had little difficulty interpreting counterexamples generated for abstractions of SCR specifications, because they are couched in terms of variables in the original specification. Also, since our abstraction methods are sound and complete, a counterexample for a property will be generated for the abstract machine if and only if the property does not hold for the original machine. We view these as important advantages of our abstraction methods.

We recently applied our abstraction methods to a simplified subset of the bomb release requirements of a U.S. Navy attack aircraft [1]. The SCR requirements specification of this system describes conditions under which the aircraft's OFP is required to issue a bomb release pulse. This specification, called `Bombrel`, contains several seeded errors. In addition to uncovering all the seeded errors with other tools in our toolset, we also established by model checking that the original formulation of a presumed one-state invariant, "*The aircraft should not drop a bomb unless the pilot has pressed release enable* (property $\mathcal{P}$ in Table 2)," does not hold for the corrected SCR specification. In consultation with Kirby, the creator of the specification, we reformulated the property as a two-state invariant (property $\mathcal{Q}$ in Table 1) and verified the restated property using both Spin and SMV.

We ran these experiments on a lightly loaded 167 MHz Sparc Ultra-1 with 130 MBytes of RAM. We used Spin Version 2.9.7 of April 18, 1997, and SMV r2.4 of December 16, 1994, in our experiments. Our tool generated the *Promela* code automatically from the SCR requirements specifications. The first abstraction method was applied automatically, while the second method was applied manually. The abstract models produced were then analyzed automatically by the toolset using Spin. Generation and analysis of the SMV model were carried out manually. (Both the process of translating an SCR specification into SMV and the application of the second abstraction method are being automated.)

6.1 Discussion

Table 1 presents some of our verification results. In Table 1, AM1 and AM2 refer to the two abstraction methods. The symbol '∞' in the table means that the corresponding model checker ran out of memory before its evaluation of the given property was complete. Spin does better than SMV (in terms of space and time requirements) for the complete SIS specification. However,

Verifying Properties With Spin

Specification	Property	AM1	AM2	States	Time	Memory
SIS	1 or 2			459,084	9s	16 MBytes
SIS	1	$\surd$		459,084	10s	16 MBytes
SIS	1 or 2		$\surd$	160	0s	3.1 MBytes
SIS	1	$\surd$	$\surd$	160	0s	3.1 MBytes
Bombrel	$\mathcal{Q}$			∞	–	–
Bombrel	$\mathcal{Q}$		$\surd$	148,354	2s	6.2 MBytes

Verifying Properties With SMV

Specification	Property	AM1	AM2	BDD Nodes	Time	Memory
SIS	1 or 2			44,653	308s	34 MBytes
SIS	1	$\surd$		44,648	309s	34 MBytes
SIS	1 or 2		$\surd$	314	0s	0.9 MBytes
SIS	1	$\surd$	$\surd$	251	0s	0.9 MBytes
Bombrel	$\mathcal{Q}$			∞	–	–
Bombrel	$\mathcal{Q}$		$\surd$	1,912	0s	0.9 MBytes

Table 1: Verifying SCR Specifications with Model Checkers.

SMV does somewhat better than Spin on the abstraction. As we expected, both Spin and SMV consume much less space and time on the abstraction than on the complete specification. For `Bombrel`, both Spin and SMV ran out of space during model checking. As in the SIS example, both Spin and SMV are able to model check an abstraction of `Bombrel`, and SMV does slightly better than Spin.

Table 2 shows that our abstraction methods dramatically reduce the time and space requirements for counterexample generation. Moreover, the generated counterexamples are significantly shorter, and therefore more easily understood. For example, an initial run of Spin on an abstraction of `Bombrel` produced a counterexample with 104 states. The shortest counterexample produced by Spin had 25 states. After examining this counterexample, Kirby manually shortened the counterexample to 9 states. SMV, however, did better by producing a counterexample with only 7 states.

We note that for the safety injection example, using Abstraction Method 1 to verify Property 1 has no effect on the number of states or the memory requirement. This is not surprising since Abstraction Method 1 only eliminates a single Boolean variable `SafetyInjection` and hence the space for storing each state remains unaffected. Further, since `SafetyInjection` is a function of only the current state (recall that the table for `SafetyInjection` is a condition table), elimination of the variable does not reduce the number of reachable states.

7 Related Work

Our approach to model checking SCR requirements specifications is a generalization and extension of the approach originally formulated and further developed by Atlee and her colleagues

Generating Counterexamples With Spin

Specification	Property	AM1	AM2	Length	Time	Memory
SIS	3			6	0.1s	2.5 MBytes
SIS	3	$\surd$	$\surd$	6	0.1s	3.1 MBytes
Bombrel	$\mathcal{P}$			1,383	315s	18 MBytes
Bombrel	$\mathcal{P}$		$\surd$	25	1.1s	5 MBytes

Generating Counterexamples With SMV

Specification	Property	AM1	AM2	Length	Time	Memory
SIS	3			4	309s	34 MBytes
SIS	3	$\surd$	$\surd$	4	0s	0.9 MByte
Bombrel	$\mathcal{P}$			–	13 Hrs	?
Bombrel	$\mathcal{P}$		$\surd$	7	0.3s	1 MByte

Table 2: Detecting Errors in SCR Specifications With Model Checking.

[4, 3, 37]. The relationship between our work and Atlee's work is described above.

In [2], Anderson et al. use SMV to analyze a component of a preliminary version of the TCAS II (Traffic Alert and Collision Avoidance System) requirements specification expressed in RSML (Requirements State Machine Language). Like us, they define schemas for translating concepts underlying RSML (such as events, input variables, environment assumptions, and the synchrony hypothesis) into suitable SMV constructs. Unlike our approach, their translation also deals with hierarchical states and timing. (We have begun to support both hierarchy and timing in SCR specifications; see, e.g., [16].) Another important difference between their approach and ours is that their translation involved significant manual effort (such as modifications to SMV and the use of special-purpose macro processors). In contrast, we use both Spin and SMV "out of the box".

The most significant difference between the two approaches is in the way integer variables and constants are handled. The problem is that of state explosion—since the encoding in SMV for integer variables (and operations on them) is not optimal, the BDDs blow up, even in specifications containing just one or two integer variables. To solve this problem, Anderson et al. directly encode integer variables as BDD bits and implement addition and comparison at the source code level by defining parameterized macros which are preprocessed using *awk* scripts. In contrast, we effectively avoid the problem by applying our correctness preserving abstraction methods to specifications containing integer variables. Because we only model check the abstractions, the state spaces in our examples appear to be up to an order of magnitude smaller than the state spaces Anderson et al. analyze. In reality, the state spaces probably are comparable in size.

Our abstraction methods are most similar to those described by Clarke et al. [7] and are also related to techniques proposed by Kurshan [29]. Clarke et al. represent each state as a vector of variable values and all abstraction mappings are obtained by (simultaneous) abstraction of the variables. Rather than *variable abstraction*, we do *variable restriction*—each of our methods simply eliminates one or more of the variables in a specification. Note that variable abstraction can achieve the same effect as variable restriction if each variable to be eliminated is mapped to a single value.

For Clarke et al., any abstraction of an automaton M determines a corresponding minimal abstraction for M called M_{min}. The difficulty with using M_{min} as an abstraction is that predicates representing the initial states and the next states of M_{min} may not exist. Hence, computing the start states and the next states of M_{min} can require references to the variables of M, which is definitely something to avoid. To solve this problem, Clarke et al. define an alternative machine called M_{app} (called an *approximate machine*) with the same states as M_{min}, but whose initial state and next-state predicates can be obtained directly. It can be shown that M_{app} is an abstraction of M. Clarke et al. use a concretization function C to transform any property ϕ_A of M_{app} into a property ϕ of M. Thus, to prove a property ϕ of M, one must choose the abstraction map h so that $C(\phi_A) = \phi$. In contrast, in performing variable restriction, we provide two general and automatable methods for finding the abstraction map. Moreover, we construct the abstract property ϕ_A from the original property ϕ rather than the other way around.

Clarke et al. do not consider complete abstractions for a given machine and property; the only version of completeness that they consider is *exactness*—simultaneous completeness for all "interesting" properties of the machine, i.e., all properties that can be expressed in terms of the primitive predicates of the machine. In contrast to our notion of completeness, exactness is not useful in practice, because the number of states in the reduced machine is only marginally smaller than the number in the original machine. Instead, our abstraction methods can dramatically reduce the number of states.

Kurshan [29] generally deals with homomorphic reductions (i.e., many-to-one mappings) in terms of the languages accepted by automata. However, he describes an equivalent notion of homomorphism between automata, which he calls "state homomorphism", that is analogous to our notion of abstraction. For state homomorphisms, Kurshan has notions of sound and complete ("exact") abstractions analogous to ours. However, where we represent automata as state machines and properties abstractly as state predicates, Kurshan represents automata as labeled transition systems and properties of an automaton as acceptable execution sequences (which can be viewed as a language of infinite strings of transition labels). Hence, while Kurshan's approach resembles ours at the abstract level, at the detailed technical level, his approach is very different and more complex than ours.

8 Conclusions

This paper has presented an approach based on the formal requirements model defined in [18] for model checking complete SCR requirements specifications with a variety of variable types for one-state and two-state invariants, the two classes of properties commonly found in specifications of practical systems. The paper also proposes two abstraction methods that make the analysis of SCR specifications practical and techniques for translating SCR specifications into the explicit state model checker, Spin, and the symbolic model checker, SMV. Finally, the paper presents some experimental results which suggest that symbolic model checking does not always perform better than explicit state model checking in detecting errors in software specifications.

We have successfully applied our abstraction methods to a practical Navy system [17]. Our abstraction methods for SCR work well in practice primarily because SCR specifications, if written and organized in accordance with the SCR method, *already contain many useful abstractions*. Therefore, unlike other approaches where the abstractions that make verification feasible must be "reverse-engineered" from scratch, useful abstractions already exist or are easy to derive (sometimes even automatically) for well-written SCR specifications.

As noted above, the translation of SCR specifications into the restricted language of SMV is being automated. The importance of automatic translation cannot be overemphasized. Hand translation of the specifications is highly error-prone; in fact, we made some subtle mistakes that were caught because the results of model checking using Spin, where the translation was automatic, were inconsistent with the results of model checking using a manual translation to SMV.

We are extending our work in model checking SCR specifications in several ways:

- We are developing additional abstraction methods.
- We are developing formal underpinnings for our abstraction methods. In particular, we have shown formally that the two abstractions methods described above are both sound and complete for "SCR machines," the state machines described by the SCR requirements model.
- We are designing algorithms to implement our abstraction methods: these algorithms automatically extract the abstraction Σ_A and the property q_A from the original SCR specification and a given property q. (In the new abstraction methods that we are developing, q_A may not be syntactically equivalent to q.)
- We also are investigating the extent to which we can automatically check that the conditions for completeness described in Section 4.2 are satisfied.
- Finally, we are developing software that will automatically translate any counterexample produced by model checking the abstraction Σ_A into a corresponding counterexample in the original specification. Currently, we perform this translation manually; whether we can produce "natural" counterexamples in a completely automatic way is an open question.

Our long-term goal is to combine the power of theorem proving technology with the ease of use of model checking technology. The major problem with current theorem proving technology, e.g., PVS [33] and ACL2 [27, 41], is that applying the technology requires mathematical sophistication and theorem proving skills. The major problem with model checking is state explosion. Clearly, theorem proving, in many cases, *automatic* theorem proving, can dramatically reduce the number of states that a model checker analyzes. Our abstraction methods are mathematically sound methods that can dramatically reduce the state space by eliminating information irrelevant to the property of interest and abstracting away unneeded detail. We are also exploring other automated techniques that address the state explosion problem, including the automatic generation of invariants and the use of powerful decision procedures.

To date, our requirements model has provided a solid foundation for a suite of analysis tools which can detect errors automatically and make the cause of those errors understandable, thereby facilitating error correction. Such an approach should lead to the production of high quality requirements specifications, which should in turn produce systems that are more likely to perform as required and less likely to lead to accidents. Such high-quality specifications should also lead to significant reductions in software development costs.

Acknowledgments

We gratefully acknowledge Myla Archer, who identified the conditions sufficient for completeness of the second abstraction method and who helped us clarify the relationship between our abstraction methods and alternative abstraction methods. We are also very grateful to R. Jeffords, who detected a subtle error in our translation of the Inmode construct into SMV, and who pointed out an anomaly in the SMV semantics. We further thank Myla and Ralph for many helpful discussions and for their excellent comments on this paper. We also thank Michal Iglewski for his constructive comments on this paper. Finally, we thank Todd Grimm and Bruce Labaw for implementing the first abstraction method and for automating the translation of SCR specifications into *Promela*.

References

[1] Thomas A. Alspaugh, Stuart R. Faulk, Kathryn Heninger Britton, R. Alan Parker, David L. Parnas, and John E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Lab., Wash., DC, 1992.

[2] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. In *Proc. Fourth ACM SIGSOFT Symp. on Foundations of Software Engineering*, October 1996.

[3] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR specifications. In *Proc. Int'l Symposium on Software Testing and Analysis*, January 1996.

[4] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng.*, 19(1):24–40, January 1993.

[5] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19, 1992.

[6] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.

[7] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc., Principles of Programming Languages (POPL)*, 1994.

[8] E. M. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang. and Systems*, 8(2):244–263, April 1986.

[9] P.-J. Courtois and David L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD, 1993.

[10] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[11] Stuart R. Faulk, John Brackett, Paul Ward, and James Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[12] Stuart R. Faulk, Lisa Finneran, James Kirby, Jr., S. Shah, and J. Sutton. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94)*, pages 3–8, Gaithersburg, MD, June 1994.

[13] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, pages 176–185, 1990.

[14] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[15] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 109–122, Gaithersburg, MD, June 1995.

[16] Constance Heitmeyer, James Kirby, and Bruce Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.

[17] Constance Heitmeyer, James Kirby, Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using model checking and simulation to detect a safety violation in a control system specification. Submitted for publication.

[18] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical Report NRL-7499, Naval Research Lab., Wash., DC, 1997. In preparation.

[19] Kathryn Heninger, David L. Parnas, John E. Shore, and John W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[20] S. D. Hester, David L. Parnas, and D. F. Utter. Using documentation as a software design medium. *Bell System Tech. J.*, 60(8):1941–1977, October 1981.

[21] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[22] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997.

[23] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. FORTE94*, October 1994.

[24] D. Jackson. Model checking and requirements, January 1997. Minitutorial.

[25] D. Jackson, S. Jha, and C.A. Damon. Faster checking of software specifications using isomorphs. In *Proc., Principles of Programming Languages (POPL)*, 1994.

[26] Ralph Jeffords. Translating SCR properties into LTL and CTL, 1997. Technical Memorandum 5540-293A:rdj.

[27] M. Kaufmann and J S. Moore. An industrial-strength theorem prover based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[28] James Kirby. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies, 1987.

[29] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[30] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[31] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[32] S. Meyer and Stephanie White. Software requirements methodology and tool study for A6-E technology transfer. Technical report, Grumman Aerospace Corp., Bethpage, NY, July 1983.

[33] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[34] David L. Parnas, G.J.K. Asmis, and Jan Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April–June 1991.

[35] David L. Parnas and Jan Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[36] Scott T. Probst. *Chemical Process Safety and Operability Analysis using Symbolic Model Checking.* Phd thesis, Carnegie-Mellon University, Department of Chemical Engineering, Pittsburgh, PA, 1996.

[37] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements. In *Proc. 11th Annual Conference on Computer Assurance (COMPASS '96)*, Gaithersburg, MD, June 1996.

[38] J. X. Su, D. L. Dill, and C. W. Barrett. Automatic generation of invariants in processor verification. In *Proc. FMCAD'96, Int'l Conference on Formal Methods in Computer-Aided Design*, November 1996.

[39] James Sutton, September 1997. Personal communication.

[40] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, pages 156–165, 1990.

[41] W.D. Young. Comparing verification systems: Interactive consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, April 1997.

A Specifying a Simple Control System in SCR

The system, a simplified version of a control system for safety injection [9], monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold. The system operator may block this process by pressing a "Block" switch. The system is reset by a "Reset" switch. To specify the requirements of the control system, we use the monitored variables `WaterPres`, `Block`, and `Reset` to denote monitored quantities, and a controlled variable `SafetyInjection` to denote the controlled quantity. The specification includes a mode class `Pressure`, a term `Overridden`, and several conditions and events.

The mode class `Pressure`, an abstract model of `WaterPres`, has three modes: `TooLow`, `Permitted`, and `High`. At any given time, the system must be in one and only one of these modes. A drop in water pressure below a constant `Low` causes the system to enter mode `TooLow`; an increase in pressure above a larger constant `Permit` causes the system to enter mode `High`. Table 3 is a mode transition table which specifies the mode class `Pressure`. In this example, the constants `Low` and `Permit` are assigned the values 900 and 1000.

Old Mode	**Event**	**New Mode**
`TooLow`	@T(`WaterPres` $\geq$ `Low`)	`Permitted`
`Permitted`	@T(`WaterPres` $\geq$ `Permit`)	`High`
`Permitted`	@T(`WaterPres` $<$ `Low`)	`TooLow`
`High`	@T(`WaterPres` $<$ `Permit`)	`Permitted`

Table 3: Mode Transition Table for `Pressure`.

The term `Overridden` is *true* if safety injection is blocked, and *false* otherwise. Table 4 is an event table which specifies the behavior of `Overridden`. The expression "`@T(Inmode)`" in a row of an event table denotes the event "system enters the corresponding mode". For instance, the entry in the first row of Table 4 specifies the event "the system enters mode `High`".

Mode	**Events**	
`High`	False	@T(Inmode)
`TooLow`, `Permitted`	@T(`Block=On`) WHEN `Reset=Off`	@T(Inmode) OR @T(`Reset=On`)
`Overridden`	True	False

Table 4: Event Table for `Overridden`.

Table 5 is a condition table that specifies the controlled quantity `SafetyInjection`. The table states that "If `Pressure` is `High` or `Permitted` *or* if `Pressure` is `TooLow` and `Overridden` is *true*, then `SafetyInjection` is `Off`; otherwise, it is `On`".

Mode	**Conditions**	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
Safety Injection	Off	On

Table 5: Condition Table for Safety Injection.

By applying the definitions in [18] to Tables 3–5, we obtain the following table functions for the mode class Pressure, the term Overridden, and the controlled variable SafetyInjection:

$$\texttt{Pressure}' = F_4(\texttt{Pressure}, \texttt{WaterPres}, \texttt{WaterPres}'\} =$$

$$\begin{cases} \texttt{TooLow} & \text{if} \quad \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres}' < \texttt{Low} \wedge \\ & \qquad \texttt{WaterPres} \not< \texttt{Low} \\ \texttt{High} & \text{if} \quad \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres}' \geq \texttt{Permit} \wedge \\ & \qquad \texttt{WaterPres} \not\geq \texttt{Permit} \\ \texttt{Permitted} & \text{if} \quad (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{WaterPres}' \geq \texttt{Low} \wedge \\ & \qquad \texttt{WaterPres} \not\geq \texttt{Low}) \vee \\ & \qquad (\texttt{Pressure} = \texttt{High} \wedge \texttt{WaterPres}' < \texttt{Permit} \wedge \\ & \qquad \texttt{WaterPres} \not< \texttt{Permit}) \\ \texttt{Pressure} & \text{otherwise.} \end{cases}$$

$$\texttt{Overridden}' = F_5(\texttt{Block}, \texttt{Reset}, \texttt{Pressure}, \texttt{Overridden}, \texttt{Block}', \texttt{Reset}', \texttt{Pressure}'\} =$$

$$\begin{cases} \textit{true} & \text{if} \quad (\texttt{Block}' = \texttt{On} \wedge \texttt{Block} = \texttt{Off} \wedge \\ & \qquad \texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Reset} = \texttt{Off}) \vee \\ & \qquad (\texttt{Block}' = \texttt{On} \wedge \texttt{Block} = \texttt{Off} \wedge \\ & \qquad \texttt{Pressure} = \texttt{Permitted} \wedge \texttt{Reset} = \texttt{Off}) \\ \textit{false} & \text{if} \quad (\texttt{Reset}' = \texttt{On} \wedge \texttt{Reset} = \texttt{Off} \wedge \\ & \qquad \texttt{Pressure} = \texttt{TooLow}) \vee \\ & \qquad (\texttt{Reset}' = \texttt{On} \wedge \texttt{Reset} = \texttt{Off} \wedge \\ & \qquad \texttt{Pressure} = \texttt{Permitted}) \vee \\ & \qquad (\texttt{Pressure}' = \texttt{High} \wedge \texttt{Pressure} \neq \texttt{High}) \vee \\ & \qquad ((\texttt{Pressure}' = \texttt{Permitted} \vee \texttt{Pressure}' = \texttt{TooLow}) \wedge \\ & \qquad\quad \neg(\texttt{Pressure} = \texttt{Permitted} \vee \texttt{Pressure} = \texttt{TooLow})) \\ \texttt{Overridden} & \text{otherwise} \end{cases}$$

SafetyInjection =

$$F_6(\texttt{Pressure}, \texttt{Overridden}) = \begin{cases} \texttt{Off} & \text{if} \;\; \texttt{Pressure}=\texttt{High} \vee \texttt{Pressure}=\texttt{Permitted} \vee \\ & \qquad (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \mathit{true}) \\ \texttt{On} & \text{if} \;\; \texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \mathit{false} \end{cases}$$

B *Promela* code for safety injection

```
/* This file contains the PROMELA/spin version of an SCRTool specification. */
/* It is created by SCRTool and automatically fed to Xspin. */
/* However, this file was left in the file sis.spin */
/* for you to use, look at, etc. */



/*****************************/
/*     numeric constants     */
/*****************************/
bool TRUE = 1;
bool FALSE = 0;
#define TooLow 0
#define Permitted 1
#define High 2
#define On 0
#define Off 1
#define Low 900
#define Permit 1000


/*********************************/
/*     variable declarations     */
/*********************************/
byte Block = Off;
byte BlockP = Off;
bool Overridden = FALSE;
bool OverriddenP = FALSE;
byte Reset = On;
byte ResetP = On;
byte SafetyInjection = On;
byte SafetyInjectionP = On;
int WaterPres = 14;
int WaterPresP = 14;
byte Pressure = TooLow;
byte PressureP = TooLow;
```

```
/***********************/
/*    init function    */
/***********************/

init {

  /*****************************/
  /*    main processing loop    */
  /*****************************/
  do
  ::

    /*******************************/
    /*     specification asserts     */
    /*******************************/
    /* (Reset = On AND Pressure = TooLow) => SafetyInjection = On */
    assert((!((Reset == On) && (Pressure == TooLow))) || (SafetyInjection == On));

    /**************************************************************/
    /*     simulation of monitored variable changes; do one each pass     */
    /**************************************************************/
    if
    ::if
      /* randomly select any value except the current one */
      :: (Block != On) -> BlockP = On ;
      :: (Block != Off) -> BlockP = Off ;
      fi
    ::if
      /* randomly select any value except the current one */
      :: (Reset != On) -> ResetP = On ;
      :: (Reset != Off) -> ResetP = Off ;
      fi
    ::if
      /* randomly jump to any value within the legal range of the variable */
      :: ((WaterPres + 1) <= 2000) -> WaterPresP = WaterPres + 1 ;
      :: ((WaterPres - 1) >= 0) -> WaterPresP = WaterPres - 1 ;
      :: ((WaterPres + 2) <= 2000) -> WaterPresP = WaterPres + 2 ;
      :: ((WaterPres - 2) >= 0) -> WaterPresP = WaterPres - 2 ;
      :: ((WaterPres + 3) <= 2000) -> WaterPresP = WaterPres + 3 ;
      :: ((WaterPres - 3) >= 0) -> WaterPresP = WaterPres - 3 ;
      :: ((WaterPres + 4) <= 2000) -> WaterPresP = WaterPres + 4 ;
      :: ((WaterPres - 4) >= 0) -> WaterPresP = WaterPres - 4 ;
      :: ((WaterPres + 5) <= 2000) -> WaterPresP = WaterPres + 5 ;
      :: ((WaterPres - 5) >= 0) -> WaterPresP = WaterPres - 5 ;
      :: ((WaterPres + 6) <= 2000) -> WaterPresP = WaterPres + 6 ;
      :: ((WaterPres - 6) >= 0) -> WaterPresP = WaterPres - 6 ;
      :: ((WaterPres + 7) <= 2000) -> WaterPresP = WaterPres + 7 ;
      :: ((WaterPres - 7) >= 0) -> WaterPresP = WaterPres - 7 ;
      :: ((WaterPres + 8) <= 2000) -> WaterPresP = WaterPres + 8 ;
      :: ((WaterPres - 8) >= 0) -> WaterPresP = WaterPres - 8 ;
  :: ((WaterPres + 9) <= 2000) -> WaterPresP = WaterPres + 9 ;
  :: ((WaterPres - 9) >= 0) -> WaterPresP = WaterPres - 9 ;
  :: ((WaterPres + 10) <= 2000) -> WaterPresP = WaterPres + 10 ;
  :: ((WaterPres - 10) >= 0) -> WaterPresP = WaterPres - 10 ;
  fi
fi;

/*******************************************************/
/*     executions of the functions in dependency order     */
/*******************************************************/

/* the PROMELA version of the Pressure function */
d_step{
if
/* modes: TooLow */
/* event: @T(WaterPres >= Low) */
:: (((!(WaterPres > Low)) && ((Pressure == TooLow) &&
   (!(WaterPres == Low)))) && (WaterPresP > Low))
   || (((!(WaterPres == Low)) && ((Pressure == TooLow) &&
   (!(WaterPres > Low)))) && (WaterPresP == Low))
       -> PressureP = Permitted;
/* modes: Permitted */
/* event: @T(WaterPres < Low) */
:: (((!(WaterPres < Low)) && (Pressure == Permitted)) && (WaterPresP < Low))
       -> PressureP = TooLow;
/* modes: Permitted */
/* event: @T(WaterPres >= Permit) */
:: (((!(WaterPres > Permit)) && ((Pressure == Permitted) &&
   (!(WaterPres == Permit)))) && (WaterPresP > Permit))
   || (((!(WaterPres == Permit)) && ((Pressure == Permitted) &&
      (!(WaterPres > Permit)))) && (WaterPresP == Permit))
       -> PressureP = High;
/* modes: High */
/* event: @T(WaterPres < Permit) */
:: (((!(WaterPres < Permit)) && (Pressure == High)) && (WaterPresP < Permit))
       -> PressureP = Permitted;
:: else skip;
fi;




/* the PROMELA version of the Overridden function */
if
/* modes: TooLow, Permitted */
/* event: @T(Block = On) WHEN Reset = Off */
:: (((!(Block == On)) && (((Pressure == TooLow) ||
    (Pressure == Permitted)) && (Reset == Off))) && (BlockP == On))
       -> OverriddenP = TRUE;
/* modes: High */
/* event: @T(Inmode) */
```

```
    :: ((!(Pressure == High)) && (PressureP == High)) -> OverriddenP = FALSE;
    /* modes: TooLow, Permitted */
    /* event: @T(Inmode) OR @T(Reset = On) */
    :: ((!((Pressure == TooLow) || (Pressure == Permitted))) &&
        ((PressureP == TooLow) || (PressureP == Permitted)))
       || (((!(Reset == On)) && ((Pressure == TooLow) ||
           (Pressure == Permitted))) && (ResetP == On)) -> OverriddenP = FALSE;
    :: else skip;
    fi;



    /* the PROMELA version of the SafetyInjection function */
    if
    /* modes:     High, Permitted */
    /* condition: TRUE */
    :: ((PressureP == High) || (PressureP == Permitted)) -> SafetyInjectionP = Off;
    /* modes:     TooLow */
    /* condition: Overridden */
    :: ((PressureP == TooLow) && OverriddenP) -> SafetyInjectionP = Off;
    /* modes:     TooLow */
    /* condition: Not Overridden */
    :: ((PressureP == TooLow) && (!OverriddenP)) -> SafetyInjectionP = On;
    fi;



    /*****************************************************************/
    /*     update each variable and mode class for this state change     */
    /*****************************************************************/
    Block = BlockP; Overridden = OverriddenP;
    Reset = ResetP; SafetyInjection = SafetyInjectionP;
    WaterPres = WaterPresP; Pressure = PressureP;
    }

  od /* end of main processing loop */

}
```

C SMV code for safety injection

```
MODULE main
VAR
  Block : {Off, On};
  Reset : {Off, On};
  WaterPres : 0..2000;
  Pressure : {TooLow, Permitted, High};
  Overridden : boolean;
  SafetyInjection : {Off, On};
DEFINE
  Low := 900;
  Permit := 1000;

ASSIGN
  init(Block) := Off;
  init(Reset) := On;
  init(WaterPres) := 14;
  init(Overridden) := 0;
  init(SafetyInjection) := On;
  init(Pressure) := TooLow;

  next(Block) := {Off, On};
  next(Reset) := {Off, On};
  next(WaterPres) := 0..2000;
  next(Pressure)   :=
               case
                 Pressure = Permitted &
                  next(WaterPres) < Low & !(WaterPres < Low) : TooLow;
                 Pressure = Permitted &
                  next(WaterPres) >= Permit & !(WaterPres >= Permit) : High;
                 Pressure = TooLow & next(WaterPres) >= Low &
                 !(WaterPres >= Low) | (Pressure = High &
                 next(WaterPres) < Permit & !(WaterPres < Permit)): Permitted;
                 1 : Pressure;
               esac;
  next(Overridden) :=
               case
                 Pressure = TooLow & next(Block) = On &
                 Block = Off & Reset = Off |
                 (Pressure = Permitted & next(Block) = On &
                  Block = Off & Reset = Off) : 1;
                 Pressure = TooLow & next(Reset) = On & Reset = Off |
                 (Pressure = Permitted & next(Reset) = On & Reset = Off) |
                 next(Pressure) = High & !(Pressure = High) |
                 (next(Pressure) = TooLow | next(Pressure) = Permitted) &
                 !(Pressure = Permitted | Pressure = TooLow) : 0;
                 1: Overridden;
               esac;
  next(SafetyInjection) :=
               case
                 next(Pressure) = High | next(Pressure) = Permitted |
                 (next(Pressure) = TooLow & next(Overridden)) : Off;
                 next(Pressure) = TooLow & !next(Overridden) : On;
               esac;

TRANS
  ((next(WaterPres) - WaterPres >= 1 & next(WaterPres) - WaterPres <= 10 |
    WaterPres - next(WaterPres) >= 1 & WaterPres - next(WaterPres) <= 10) &
  next(Block) = Block & next(Reset) = Reset) |
  (next(WaterPres) = WaterPres & !(next(Block) = Block) &
   next(Reset) = Reset) |
  (next(WaterPres) = WaterPres & next(Block) = Block &
   !(next(Reset) = Reset))

SPEC
 AG((Reset = On & !(Pressure = High)) -> !Overridden)
-- AG((Reset = On & Pressure = TooLow) -> SafetyInjection = On)
-- AG((Block = Off & Pressure = TooLow) -> SafetyInjection = On)
--   AG((!(Pressure = TooLow) & Block = Off) ->
--      AX(Pressure = TooLow -> SafetyInjection = On))
```